MASSACHUSETTS INSTITUTE OF TECHNOLOGY

VLSI Memo No. 88-470
August 1988

MICRO-OPTIMIZATION OF FLOATING-POINT OPERATIONS

William J. Dally

Abstract

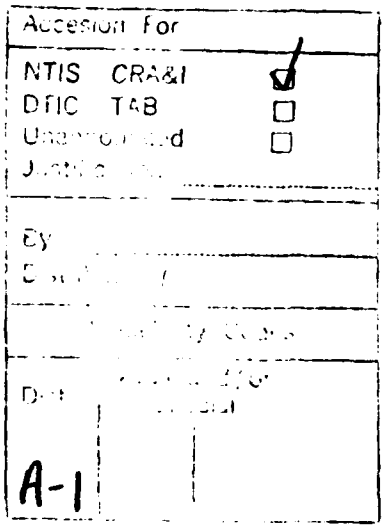
This paper describes micro-optimization, a technique for reducing the operation count and time required to perform floating-point calculations. Micro optimization involves breaking floating-point operations into their constituent micro-operations and optimizing the resulting code. Exposing micro-operations to the compiler creates many opportunities for optimization. Redundant normalization operations can be eliminated or combined. Also, scheduling micro-operations separately results allows dependent operations to be partially overlapped. A prototype expression compiler has been written to evaluate a number of micro-optimizations. On a set of benchmark expressions operation count is reduced by 33% and execution time is reduced by 40%.

Microsystems Research Center
Room 39-321

Massachusetts Institute of Technology

Cambridge Massachusetts 02139

Telephone (617) 253-8138



Acknowledgements

This work was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-80-C-0622, N00014-87-K-0825 and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigators Award with matching funds from General Electric Corporation and IBM Corporation.

Author Information

Dally: Department of Electrical Engineering and Computer Science, Artificial Intelligence Laboratory, MIT, Room NE43-417, Cambridge, MA 02139, (617) 253-6043.

Micro-Optimization of Floating-Point Operations[1]

William J. Dally
Artificial Intelligence Laboratory and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

This paper describes micro-optimization, a technique for reducing the operation count and time required to perform floating-point calculations. Micro optimization involves breaking floating-point operations into their constituent micro-operations and optimizing the resulting code. Exposing micro-operations to the compiler creates many opportunities for optimization. Redundant normalization operations can be eliminated or combined. Also, scheduling micro-operations separately results allows dependent operations to be partially overlapped. A prototype expression compiler has been written to evaluate a number of micro-optimizations. On a set of benchmark expressions operation count is reduced by 33 % and execution time is reduced by 40 %.

1 Introduction

Many unneeded operations are performed during the evaluation of floating point expressions because existing compilers and floating point units consider these operations to be atomic. By decomposing floating point operations into their constituent integer micro-operations, many opportunities for optimization are exposed. Redundant shift operations may be eliminated, parts of the computation may be done with a block exponent, common subexpressions in the mantissa or exponent calculation are exposed, and additional flexibility in scheduling operations is possible.

This paper describes methods for micro-optimizing floating point expressions. Each operation in the expression is decomposed into its primitive integer micro-operations. For example a floating point add is decomposed into an exponent subtract, mantissa alignment, mantissa add, leading zero's count, exponent adjust, and mantissa normalization. Optimizations are performed on the resulting micro-operations. For example, a normalizing left shift from one FP add may be combined with the aligning right shift of a subsequent FP add resulting in a single shift. The entire expression is scheduled as a unit resulting in better hardware utilization.

On a set of benchmark expressions, micro-optimization reduces operation count by 33 % and execution time by 40 % compared to conventional floating point execution with identical

[1]The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-80-C-0622 and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and IBM Corporation.

Submitted to ASPLOS '88

function unit performance and register bandwidth.

To fully exploit micro-optimization, a micro floating point unit (μFPU) is required. The instruction set of a μFPU consists of the micro-operations required for floating point arithmetic (e.g., alignment shifts that maintain guard, round, and sticky bits). These operations are performed out of a set of mantissa and exponent registers. By providing the appropriate primitive operations, no comprimises are made in terms of accuracy, rounding, adherence to standards, and performance.

This work is motivated by recent progress on RISC [8] and VLIW [3] architectures. RISC machines eliminate the complex addressing modes found in CISC machines [9]. Address calculations are performed using integer arithmetic instructions rather than by microcode or special hardware. Exposing these calculations to the compiler often improves performance. Micro optimization applies this technique to floating point operations. As with address calculations, breaking these operations into their primitive components has the disadvantage of decreasing code density and increasing instruction bandwidth.

Micro-optimization borrows from VLIW technology, in that several micro-operations may be performed simultaneously. Also, some of the optimizations described here schedule code across basic blocks. However, the technique used is different from trace scheduling.

The idea of using a compiler to optimize a function normally considered a primitive arithmetic operation has been applied to integer multiplication by a constant [5].

The next section illustrates the basic concepts of micro-optimization by means of a few simple examples. A prototype expression compiler written to test these concepts is described in Section 3. Section 4 describes the architecture of an exemplary μFPU. The compiler and μFPU are evaluated on a number of benchmark programs in Section 5.

2 Micro-Optimizations

This section illustrates micro-optimizations by means of examples given in μFP assembly code (see Section 4). The code for a single add (`A = B + C`) and a single multiply (`A = B * C`) are shown below. The subtract operation is similar to add. The optimizations start from concatenations of these sequences and perform transformations to reduce the number of micro-operations.

```
                ADD                                  MULTIPLY

L0:     E0  = EB  E-  EC , BNEG L1       L0:     E1  = EB  E+  EC
        M0  = MC  SHR E0                         M1  = MB  M*  MC
        M1  = M0  M+  MB , BR L2                 E2  = FF1 M1
L1:     M0  = MB  SHR- E0                        EA  = E1  E-  E2
        M1  = MB  M+  M0                         MA  = M1  SHL E2
L2:     E1  = FF1 M1
        EA  = EB  E-  E1
        MA  = M1  SHL E1
```

In this section optimizations will be evaluated by comparing the path lengths of the optimized and unoptimized μFP code. Timings for different micro-operations will be discussed in Section 4.

Three instructions, at least half the total, in each sequence are used to normalize the result. Many of the optimizations described below are methods to eliminate unneccessary normalizations.

Automatic Block Exponent

The alignment operations of cascaded additions can be simplified if the largest exponent is identified and used as a block exponent for the additions. All mantissas are aligned using this exponent and added without normalization. Only the final sum is normalized.

The following code shows an application of this technique to the expression (`TO = A + B + C`). Only the code for the case where `A` has the largest exponent is shown. By eliminating the normalization and realignment of the intermediate result, this path through the sum has been reduced from 12 instructions to 9.

```
LO:     E1  = EA  E-  EB , BNEG L1
        E2  = EA  E-  EC , BNEG L2
        M1  = MB  SHR E1
        M2  = MA  M+  M1
        M3  = MC  SHR E2
        M4  = M2  M+  M3 , BR L4
        <L1 and L2 omitted for clarity>
L4:     E4  = FF1 M1
        ETO = EA  E-  E4
        MTO = M4  SHL E4
```

The use of automatic block exponent requires that extra mantissa bits to the left of the binary point be maintained in case the adds result in an increased exponent. If n adds are performed in sequence, $\log_2 n$ extra bits must be maintained.

In some cases, the use of an automatic block exponent can increase rounding errors. In the above example, if `A` $\approx$ `-B` and `|C|` << `|A|`, the intermediate result is badly undernormalized and valuable bits of `C` will be lost when it is aligned with the original exponent. The effect is the same as if the addition were performed in the order `(A + C + B)`. This technique treats floating point addition as if it were associative and commutative and has the same effect as reordering the additions to give the largest possible rounding error.

Even with these limitations, automatic block exponent is a very effective optimization. Many computations include long sequences of adds (e.g., dot products) where operand ordering is not critical. In these cases, the use of a block exponent reduces the path length by from $6n$ to $3n + 3$, a savings of 50%!

Shift Combining

Shift combining is an alternative to automatic block exponent that can be used in cases where the order of the operations must be preserved. When adding three or more floating point numbers, redundant shifts may be performed when a mantissa is shifted left for normalization and then immediately shifted right for alignment. To recognize redundant shifts, the mantissa left shift in the first add is moved below the branch of the second add. This requires copying the shift into both paths of the branch. The shift will be eliminated in one of the two paths.

The following code fragment, taken from the compilation of `A + B + C`, illustrates this technique. The fragment begins after the B and C mantissas have already been aligned and added. It ends after the final mantissa sum is computed but before the normalization.

```
BEFORE OPTIMIZATION                    AFTER OPTIMIZATION

L2:     E1  = FF1 M1                   L2:     E1  = FF1 M1
        ET0 = EB  E-  E1                       ET0 = EB  E-  E1
        MT0 = M1  SHL E1                       E2  = EA  E-  ET0, BNEG L3
        E2  = EA  E-  ET0, BNEG L3             E3  = EA  E-  EB
        M2  = MT0 SHR E2                       M2  = M1  SHR E3
        M3  = M2  M+  MA,  BR L4               M3  = M2  M+  MA,  BR L4
L3:     M2  = MA SHR- E2               L3:     MT0 = M1  SHL E1,
        M3  = MT0  M+  M2                      M2  = MA  SHR- E2
                                               M3  = MT0 M+  M2
```

The left shift of `M1` has been pushed below the branch on (`EA >= ET0`). If the branch is not taken, the shift is combined with the alignment right shift. An additional exponent subtract is required to calculate the shift count. If the branch is taken, the shifts operate on different mantissas and cannot be combined. The path length of the optimized code is unchanged, but an expensive mantissa shift is replaced with an inexpensive exponent subtract.

Post Multiply Normalization

A multiply operation can denormalize its result by at most one bit position. If a few extra guard bits to the right of the mantissa are maintained, the results of multiplication can be used without normalization with no loss of accuracy. Only the final result must be normalized. For example, the code for `A * B * C` is shown below.

```
L0:     E1  = EB  E+  EC
        M1  = MB  M*  MC
        E2  = E1  E+  EA
        M2  = MA  M*  M1
        E3  = FF1 M2
        ET0 = E3  E-  E2
        MT0 = M2  SHL E3
```

This optimization also handles the ubiquitous case of multiply-add. If a multiply is followed by an add, its normalization can be eliminated as the final result will be normalized by the add.

For a sequence of multiplies, this optimization reduced the number of intructions from $5n$ to $2n + 3$, a savings of 60%. The savings in terms of time is somewhat less since the mantissa multiply M* is an extremely costly operation.

Conventional Optimizations

Decomposing floating-point operations exposes the resulting micro-operations to conventional compiler optimizations such as constant folding, common subexpression elimination, and dead code elimination. Consider for example, the expression (A + B)*(A - B). When reduced to micro-operations the alignment of A and B can be recognized as a common subexpression and eliminated. The optimization reduces the path length from 17 to 15, a 12% improvement. A source level compiler can find no common subexpressions and will perform the alignment twice.

Scheduling

More efficient use of floating point hardware can be made by scheduling the micro-operations of an entire floating-point expresion as a unit rather than scheduling each add or multiply separately. The μops of one floating point operations can be used to fill idle cycles in the evaluation of other floating point operations even if there are dependencies between the two operations.

Consider for example the case of a multiply-add (A * B + C). A reservation table for this operation is shown below. Once the exponent addition for the multiply is completed (A), the exponent subtract for the add may be performed (C). If EA + EB > EC, the alignment shift for the add (D) may also be performed in parallel with the multiply (B). In a conventional floating point unit, the multiply has to complete before any part of the add can be performed.

```
Unit     1 2 3 4 5 6 7 8 9 10
M *      B B B B
M +              E E
M SH           D D           H H
M FF1                    F F
E +/-    A C                 G
```

```
A: E1  = EA  E+  EB
B: M1  = MA  M*  MB
C: E2  = E1  E-  EC, BNEG L1
D: M2  = MC  SHR E2
E: M3  = M1  M+  M2, BR   L2
```

3 The Micro-Optimizer

An experimental micro-optimizer has been implemented to evaluate the optimizations described above. The program accepts a restricted LISP expression as input and produces optimized μFPU assembly code as output.

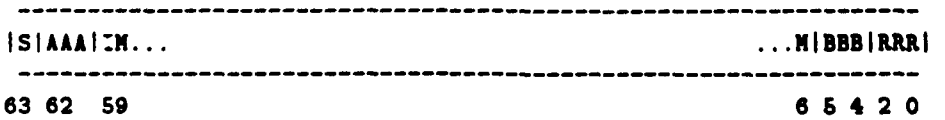
The compilation is performed in the following steps

1. The expression is compiled into standard three address *macro* floating-point assembly code.
2. A data flow graph is constructed and used to recognize (1) sequences of cascaded additions and (2) non-terminal multiplies.
3. With the aid of the data flow graph, the *macro* assembly code is translated into μFP code. Automatic block exponent and post multiply normalization optmizations are performed during this step.
4. Shift combining is performed by checking each shift to determine if its result is used as input to another shift.
5. A control flow graph is constructed and each statement is labeled with an identifier specifying the paths that pass through that statement.
6. With the aid of the control flow graph, common subexpression elimination is performed. Expressions are eliminated outside of basic blocks if they are labeled with the same path identifier.
7. The optimized μFP code is scheduled into horizontal microinstructions using a greedy algorithm that schedules an operation as soon as its inputs and required resources are available.

4 A Micro Floating Point Unit

A micro floating point unit (μFP) is required to efficiently execute the code produced by the micro-optimizer. Micro-optimization reduces floating point operations to their constituent integer operations; however an integer processor does not support features such as *sticky* bits that are required to round according to existing standards [2]. This section describes the architecture of a μFP suitable to execute the code described above. The purpose of this design is to serve as a basis for the evaluation made in Section 5. This description is a paper design, no μFPU has been constructed.

The μFP contains a 31-word by 12-bit exponent register file, and a 31-word by 64-bit mantissa file. Each register file has two read ports and a single write port. The exponent registers contain 12-bit 2's complement numbers. These numbers are converted to/from offset format during load and store operations. The mantissa registers have the format shown below. A 55-bit mantissa (M) includes the implied bit (I), and sign bit S. The mantissa is protected above by three A bits and below by three B bits as well as the standard guard, round, and sticky bits (R).



The A bits allow up to four aligned mantissa additions to be performed before normalizing the result. The possible one-bit overflows are accumulated in the A bits for later normalization. The B bits allow up to four multiplies to be performed before normalizing. The bits that shift off to the right because of the possible one-bit denormalization are accumulated in the B bits and the guard bit.

The exponent and mantissa data paths are shown in Figure 1. The exponent path has an adder/subtractor and can receive data from the find-first-one (FF1) unit in the mantissa path. The mantissa path includes a multiplier, an adder, a shifter, and a find-first-one unit. The multiplier, adder, shifter, and FF1 unit are pipelined with latencies of 4,2,2, and 2 (see below). The shifter sets the sticky bit of the result if any ones are discarded from the right side of the operand. The adder uses the round and sticky bits to round each addition. The multiplier both produces the rounding bits and uses them to round the result.

There are two crossovers between the exponent and mantissa data paths. The mantissa shift is controlled by an exponent shift count, and the find-first-one unit takes a mantissa as input and produces an exponent result.

The clock cycle is determined by the time required for a 12-bit exponent add ($\approx$ 15ns in a 1μ CMOS technology). Assuming a carry lookahead adder and a Wallace-tree multiplier [4], times for mantissa multiply, add, shift, and find-first-one are estimated to be 4, 2, 2, and 2 cycles respectively. A register read or write takes one clock cycle, and a register can be read in the same cycle it is written. There is full bypassing under compiler control (no comparators).

The format of a μFP instruction is shown below. Each instruction specifies sources and destinations for the mantissa and exponent register files, the exponent and mantissa operations, and a branch specifier. Specifying a register address of all ones (0x1F) selects a bypass from the result bus. Branches have no delay if not taken and a one cycle delay if taken.

Instruction Format:

EA	EB	EC	MA	MB	MC	EOP	MOP	BOP	BDST

The units perform the following operations. Each unit also has a NOP operation.

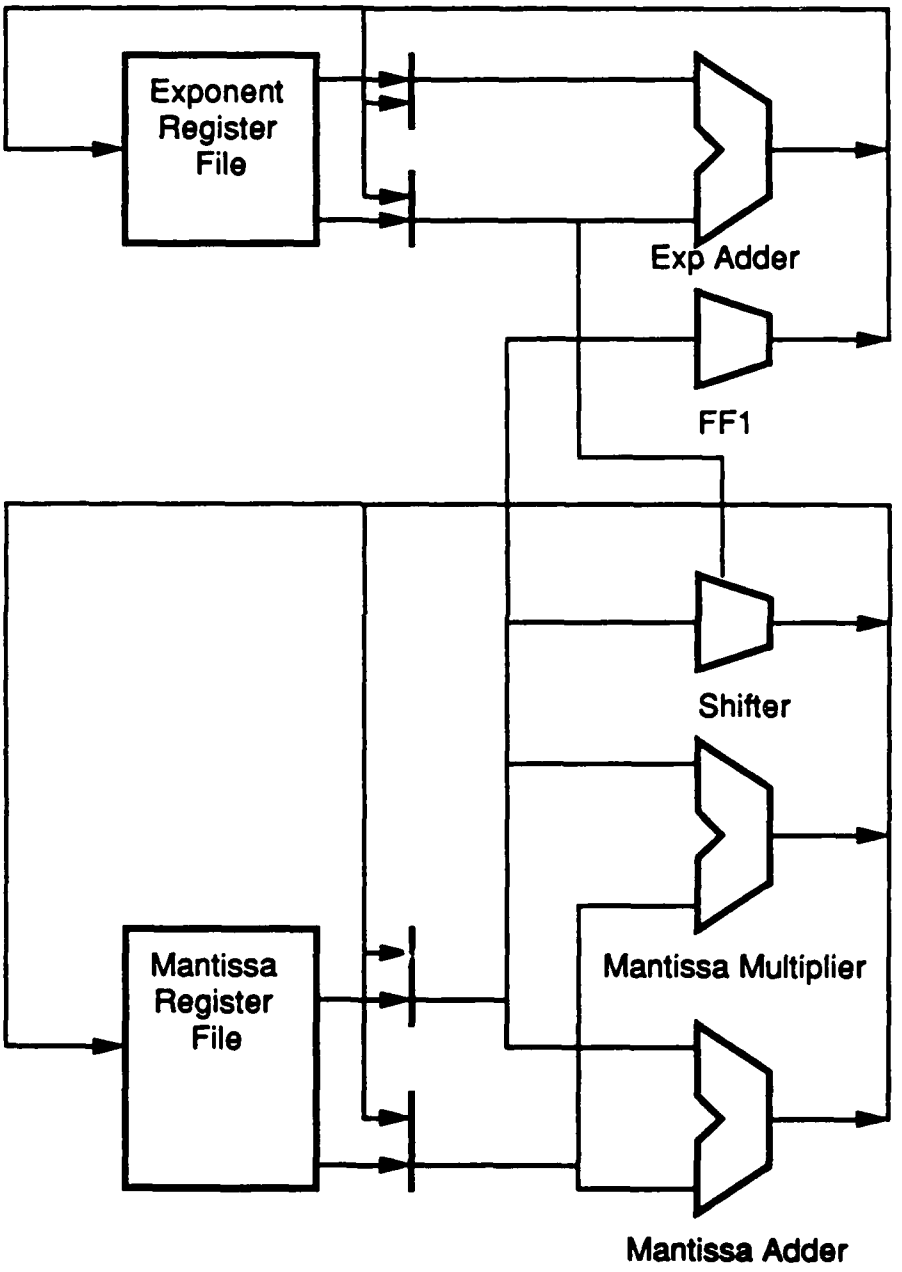


Figure 1: μFPU Data Paths

Exponent OPs	
E+, E-	Exponent add/subtract (EC <- EA op EB).
FF1	Returns the shift required to normalize mantissa MA (EC <- FF1 MA). In the range [-3,57]. Returns the largest positive number if no ones are found.
LDE, STE	Load or store exponent as an integer.
Mantissa OPs	
M+, M-, M*	Mantissa add, subtract, and multiply (MC <- MA op MB).
SHR, SHL	Mantissa right and left shift (MC <- MA >> EA) or (MC <- MA << EA). A negative exponent shifts in the opposite direction.
ABS, NEG	Zeros and complements the mantissa sign bit.
LDM, STM	Load or store mantissa as an integer.
LDF, STF	Load or store mantissa and exponent formatted as a standard floating point number.
Branch OPs BR	Unconditional branch.
BNEG	Branch on exponent negative (EC < 0).
Bcond	Branch on exponent and mantissa compare (EA, MA) relop (EB, MB).

This instruction set *is the minimum* required to perform the evaluation in the next section. In certain applications additional instructions would be useful. For example, if divides were used frequently a mantissa divide M/ could be realized with an SRT divide array. If divides are less frequent, a reciprocal approximation can be programmed using the instructions above.

This instruction set is intended to complement a simple integer instruction set [7] [1] [6]. For operations such as reciprocal and square root that are often performed using Newton's method, there is no need to implement an initial approximation lookup table in the μFPU. These tables can be kept in main memory and accessed using integer instructions. By exposing the algorithms for reciprocal, square root, and other floating-point functions, the compiler can perform optimizations that are not possible if these functions are hidden in microcode.

5 Evaluation

To evaluate micro-optimizations, the μFPU described in Section 4 is compared against a conventional floating point unit (cFPU) with the same micro-operation times and register file bandwidth. The two units were compared on a series of benchmark expressions. For each expression and each unit, the total number of micro operations *operation count* and the total number of clock cycles required *time* to execute the longest path through the expression is measured.

The following assumptions are made:

- The two units have identical clock rates and micro operation times.

- Each cycle, each unit can read two mantissas and two exponents and write one mantissa and one exponent.
- All units are pipelined and can accept a new input each cycle.
- Branches have no delay if not taken and a delay of one if taken.
- Common subexpression elimination is performed on the **macro** floating point operations for both units.
- The operations on each unit were scheduled using a greedy algorithm.

The benchmarks are summarized in the following table:

Benchmark	Description	*	+
1	(+ (* a a) (* b b)))	2	1
2	(+ a b c d))	0	3
3	(* a b c d))	3	0
4	Simple MOSFET Equation	3	3
5	3-D dot product	3	2
6	Acceleration Calculation	8	7
7	Magnitude of Butterfly	8	9
8	8 Tap FIR Filter	8	7

The operation counts and times for the twelve cases are tabulated below along with total lengths and times for the two units.

Over the six benchmarks, micro-optimizations resulted in a 33 % reduction in operation count and a 40 % reduction in time. The reductions are largest for large expressions with long sequences of adds or multiplies.

Expressions with a great deal of internal parallelism give a smaller reduction in execution time. The parallelism in these expressions can keep a conventional floating point pipeline very busy reducing the advantage gained by independently scheduling micro-operations. For example, the FFT butterfly operation (benchmark 7) calculates the real and imaginary components of its two outputs in parallel. A pipelined FPU can execute these four calculations in parallel. Because the μFPU consumes register bandwidth handling intermediate results, it cannot initiate operations as quickly. Because of the register bandwidth bottleneck, this benchmark has a typical reduction in operation count (30%), but only a 25% reduction in execution time.

All benchmarks other than number 7 show a greater improvement in execution time than in operation count. This data suggests that register bandwidth is not an issue for most scalar expressions. The two units were compared with identical and realistic register file bandwidth. Data dependencies prevent the conventional FPU from exploiting all of this bandwidth. If memory bandwidth is equal to register bandwidth, a conventional FPU will outperform a μFPU on vector operations. The conventional unit can start an operation each cycle while the

μFPU will use some register bandwidth for intermediate results. When register bandwidth is at least twice memory bandwidth, the μFPU becomes competitive even on vector operations.

Operation Count			
Benchmark	cFPU	μFPU	% Reduction
1	16	10	38
2	18	15	17
3	15	9	40
4	33	26	21
5	27	17	37
6	82	56	32
7	94	67	29
8	82	47	43
TOTAL	367	247	33

Time (cycles)			
Benchmark	cFPU	μFPU	% Reduction
1	21	13	38
2	30	19	37
3	30	16	47
4	50	31	38
5	32	17	47
6	94	52	45
7	73	55	25
8	87	47	46
TOTAL	417	250	40

6 Conclusion

A technique for micro-optimizing floating-point expressions has been described. Micro-optimization involves reducing floating-point expressions to their constituent micro-operations and optimizing the resulting sequence. By exposing the micro-operations to the compiler many redundant operations can be eliminated. Scheduling of individual micro-operations allows dependent macro operations to be partially overlapped.

An evaluation of micro-optimization shows that it reduces operation count by 33 % and execution time by 40 % compared to conventional floating-point execution. The operation count reduction is largely due to the elimination of unecessary normalization operations. Elimination of common exponent subexpressions contributes a small amount. The improvement in execution time is due to the elimination of these operations and the increased overlap of operations resulting from scheduling micro-operations separately. In some cases exponent calculations are scheduled in such a manner that the execution time is entirely due to mantissa calculations.

A micro floating-point unit is required to execute these floating-point micro-operations. Although they are integer operations, appropriate word lengths and support for rounding are required to maintain accuracy. Also, separate mantissa and exponent paths are required to give performance competitive with conventional floating point units.

A μFPU breaks the pipeline of a conventional floating-point unit into separately schedulable function units. The additional scheduling flexibility can be exploited through micro-optimization. The penalty for this separation is potentially higher register file bandwidth, higher instruction bandwidth and increased control complexity.

The flexibility inherent in a μFPU has many advantages other than performance. For example, it can be used to gracefully support high precision floating point numbers. If provision is made in the μFPU to recover the low bits of a multiply and to link carry bits between adds, high-precision floating point arithmetic can be implemented at about the same cost as high-precision integer arithmetic.

A μFPU can also make tradeoffs between area and performance. For example, a smaller unit could be constructed that performs mantissa multiply with two or four multiply step operations. The resulting unit would be significantly smaller and would be slower only in those cases where two mantissa multiplies can be overlapped.

The work described here is an effort to integrate floating-point arithmetic into RISC computer architecture [8]. Conventional RISCs operate with a scalar and/or vector floating point unit that is operated separately from the RISC pipeline. A μFPU integrates floating point operations into the pipeline so that only one execution controller is required. Floating-point micro-operations are handled in the same manner as integer operations.

Most floating point calculations are limited by memory bandwidth rather than by arithmetic capability. By integrating floating-point and address calculation in one unit, the coupling between the FPU and the memory system can be made tighter. For example, micro-operations can be used to fill the delay slots of a delayed load. Because these operations are scheduled by the compiler, no time and bandwidth is lost synchronizing data arrival with a separately scheduled floating point pipeline.

Much work remains to be done on micro-optimizations. Extending the expression compiler of Section 3 into a full compiler will create opportunities for additional optimization. For example, loops that iterate over arrays accumulating a running sum can be optimized with a technique similar to automatic block exponent. Other optimizations become possible if the compiler is extended to infer the signs and relative magnitudes of some variables. If the two inputs to a mantissa add can be shown to have the same sign, the result will not be denormalized (it may overflow one bit), and the sign of the result can be inferred. If exponent values can be inferred or computed early, block exponents can be applied across large expressions. If the relative magnitudes of exponents can be inferred, branches on exponent comparison can be eliminated.

Floating point numbers are popular because they free the programmer from the tedious task of scaling integers. Scaling need not be performed entirely at run-time by hardware, however. A suitable division of effort between a micro-optimizing compiler and hardware with some *primitive* support for floating point can result in substantial performance improvement.

Acknowledgement

The work presented here has benefited from discussions with Anant Agarwal, Tom Knight, Scott Wills, and Steve Ward.

References

[1] AMD, *AMD 29000 User's Manual*, 1987.

[2] ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic.*

[3] Colwell, R.P., et.al., "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. Computers*, C-37(8), August 1988, pp. 967-979.

[4] Hwang, K. , *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, 1979.

[5] Magenheimer, et.al., "Integer Multiplication and Division on the HP Precision Architecture," *IEEE Trans. Computers*, C-37(8), August 1988, pp. 980-990.

[6] Motorola, *MC88100 32-bit Third-Generation RISC Microprocessor: Technical Summary*, Document BR588/D, 1988.

[7] Moussouris, J. et.al, "A CMOS RISC Processor with Integrated System Function," *COMPCON*, 1986, pp. 126-131.

[8] Patterson, David A., "Reduced Instruction Set Computers," *CACM*, 28(1), January 1985, pp. 8-21.

[9] Strecker, W.D., "VAX-11/780, A Virtual Address Extension to the PDP-11 Family", Proc. NCC, 1978, pp. 967-980.